1

Introduction and Overview

Operating systems are not only regarded as a fascinating part of information technology, but are also the subject of controversial discussion among a wide public.[1] Linux has played a major role in this development. Whereas just 10 years ago a strict distinction was made between relatively simple academic systems available in source code and commercial variants with varying performance capabilities whose sources were a well-guarded secret, nowadays anybody can download the sources of Linux (or of any other free systems) from the Internet in order to study them.

Linux is now installed on millions of systems and is used by home users and professionals alike for a wide range of tasks. From miniature embedded systems in wristwatches to massively parallel mainframes, there are countless ways of exploiting Linux productively. And this makes the sources so interesting. A sound, well-established concept (UNIX) melded with powerful innovations and a strong penchant for dealing with problems that do not arise in academic teaching systems — this is what makes Linux so fascinating.

This book describes the central functions of the kernel, explains its underlying structures, and examines its implementation. Because complex subjects are discussed, I assume that the reader already has some experience in operating systems and systems programming in C (it goes without saying that I assume some familiarity with *using* Linux systems). I touch briefly on several general concepts relevant to common operating system problems, but my prime focus is on the implementation of the Linux kernel. Readers unfamiliar with a particular topic will find explanations on relevant basics in one of the many general texts on operating systems; for example, in Tanenbaum's outstanding

[1] It is *not* the intention of this book to participate in ideological discussions such as whether Linux can be regarded as a full operating system, although it is, in fact, just a kernel that cannot function productively without relying on other components. When I speak of Linux as an operating system without explicitly mentioning the acronyms of similar projects (primarily the GNU project, which despite strong initial resistance regarding the kernel reacts extremely sensitively when *Linux* is used instead of *GNU/Linux*), this should not be taken to mean that I do not appreciate the importance of the work done by this project. Our reasons are simple and pragmatic. Where do we draw the line when citing those involved without generating such lengthy constructs as *GNU/IBM/RedHat/HP/KDE/Linux*? If this footnote makes little sense, refer to `www.gnu.org/gnu/linux-and-gnu.html`, where you will find a summary of the positions of the GNU project.
After all ideological questions have been settled, I promise to refrain from using half-page footnotes in the rest of this book.

introductions ([TW06] and [Tan07]). A solid foundation of C programming is required. Because the kernel makes use of many advanced techniques of C and, above all, of many special features of the GNU C compiler, Appendix C discusses the finer points of C with which even good programmers may not be familiar. A basic knowledge of computer structures will be useful as Linux necessarily interacts very directly with system hardware — particularly with the CPU. There are also a large number of introductory works dealing with this subject; some are listed in the reference section. When I deal with CPUs in greater depth (in most cases I take the IA-32 or AMD64 architecture as an example because Linux is used predominantly on these system architectures), I explain the relevant hardware details. When I discuss mechanisms that are not ubiquitous in daily live, I will explain the general concept behind them, but expect that readers will also consult the quoted manual pages for more advice on how a particular feature is used from userspace.

The present chapter is designed to provide an overview of the various areas of the kernel and to illustrate their fundamental relationships before moving on to lengthier descriptions of the subsystems in the following chapters.

Since the kernel evolves quickly, one question that naturally comes to mind is which version is covered in this book. I have chosen kernel 2.6.24, which was released at the end of January 2008. The dynamic nature of kernel development implies that a new kernel version will be available by the time you read this, and naturally, some details will have changed — this is unavoidable. If it were not the case, Linux would be a dead and boring system, and chances are that you would not want to read the book. While some of the details will have changed, *concepts* will not have varied essentially. This is particularly true because 2.6.24 has seen some very fundamental changes as compared to earlier versions. Developers do not rip out such things overnight, naturally.

1.1 Tasks of the Kernel

On a purely technical level, the kernel is an intermediary layer between the hardware and the software. Its purpose is to pass application requests to the hardware and to act as a low-level driver to address the devices and components of the system. Nevertheless, there are other interesting ways of viewing the kernel.

- The kernel can be regarded as an *enhanced machine* that, in the view of the application, abstracts the computer on a high level. For example, when the kernel addresses a hard disk, it must decide which path to use to copy data from disk to memory, where the data reside, which commands must be sent to the disk via which path, and so on. Applications, on the other hand, need only issue the command *that* data are to be transferred. *How* this is done is irrelevant to the application — the details are abstracted by the kernel. Application programs have no contact with the hardware itself,[2] only with the kernel, which, for them, represents the lowest level in the hierarchy they know — and is therefore an enhanced machine.
- Viewing the kernel as a *resource manager* is justified when several programs are run concurrently on a system. In this case, the kernel is an instance that shares available resources — CPU time, disk space, network connections, and so on — between the various system processes while at the same time ensuring system integrity.

[2]The CPU is an exception since it is obviously unavoidable that programs access it. Nevertheless, the full range of possible instructions is not available for applications.

- Another view of the kernel is as a *library* providing a range of system-oriented commands. As is generally known, *system calls* are used to send requests to the computer; with the help of the C standard library, these appear to the application programs as normal functions that are invoked in the same way as any other function.

1.2 Implementation Strategies

Currently, there are two main paradigms on which the implementation of operating systems is based:

1. **Microkernels** — In these, only the most elementary functions are implemented directly in a central kernel — the *micro*kernel. All other functions are delegated to autonomous processes that communicate with the central kernel via clearly defined communication interfaces — for example, various filesystems, memory management, and so on. (Of course, the most elementary level of memory management that controls communication with the system itself is in the microkernel. However, handling on the system call level is implemented in external servers.) Theoretically, this is a very elegant approach because the individual parts are clearly segregated from each other, and this forces programmers to use ''clean'' programming techniques. Other benefits of this approach are dynamic extensibility and the ability to swap important components at run time. However, owing to the additional CPU time needed to support complex communication between the components, microkernels have not really established themselves in practice although they have been the subject of active and varied research for some time now.
2. **Monolithic Kernels** — They are the alternative, traditional concept. Here, the entire code of the kernel — including all its subsystems such as memory management, filesystems, or device drivers — is packed into a single file. Each function has access to all other parts of the kernel; this can result in elaborately nested source code if programming is not done with great care.

Because, at the moment, the performance of monolithic kernels is still greater than that of microkernels, Linux was and still is implemented according to this paradigm. However, one major innovation has been introduced. *Modules* with kernel code that can be inserted or removed while the system is up-and-running support the dynamic addition of a whole range of functions to the kernel, thus compensating for some of the disadvantages of monolithic kernels. This is assisted by elaborate means of communication between the kernel and userland that allows for implementing hotplugging and dynamic loading of modules.

1.3 Elements of the Kernel

This section provides a brief overview of the various elements of the kernel and outlines the areas we will examine in more detail in the following chapters. Despite its monolithic approach, Linux is surprisingly well structured. Nevertheless, it is inevitable that its individual elements interact with each other; they share data structures, and (for performance reasons) cooperate with each other via more functions than would be necessary in a strictly segregated system. In the following chapters, I am obliged to make frequent reference to the other elements of the kernel and therefore to other chapters, although I have tried to keep the number of forward references to a minimum. For this reason, I introduce the individual elements briefly here so that you can form an impression of their role and their place in the overall

concept. Figure 1-1 provides a rough initial overview about the layers that comprise a complete Linux system, and also about some important subsystems of the kernel as such. Notice, however, that the individual subsystems will interact in a variety of additional ways in practice that are not shown in the figure.

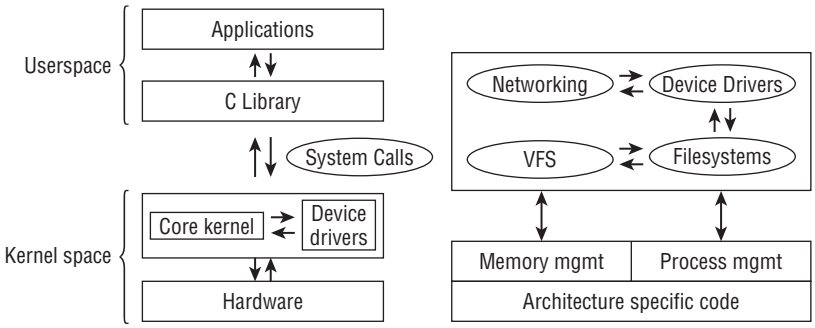

Figure 1-1: High-level overview of the structure of the Linux kernel and the layers in a complete Linux system.

1.3.1 Processes, Task Switching, and Scheduling

Applications, servers, and other programs running under UNIX are traditionally referred to as *processes*. Each process is assigned address space in the *virtual memory* of the CPU. The address spaces of the individual processes are totally independent so that the processes are unaware of each other — as far as each process is concerned, it has the impression of being the only process in the system. If processes want to communicate to exchange data, for example, then special kernel mechanisms must be used.

Because Linux is a multitasking system, it supports what appears to be concurrent execution of several processes. Since only as many processes as there are CPUs in the system can really run at the same time, the kernel switches (unnoticed by users) between the processes at short intervals to give them the impression of simultaneous processing. Here, there are two problem areas:

1. The kernel, with the help of the CPU, is responsible for the technical details of task switching. Each individual process must be given the illusion that the CPU is always available. This is achieved by saving all state-dependent elements of the process before CPU resources are withdrawn and the process is placed in an idle state. When the process is reactivated, the exact saved state is restored. Switching between processes is known as *task switching*.
2. The kernel must also decide *how* CPU time is shared between the existing processes. Important processes are given a larger share of CPU time, less important processes a smaller share. The decision as to which process runs for how long is known as *scheduling*.

1.3.2 UNIX Processes

Linux employs a hierarchical scheme in which each process depends on a parent process. The kernel starts the `init` program as the first process that is responsible for further system initialization actions and display of the login prompt or (in more widespread use today) display of a graphical login interface. `init` is therefore the root from which all processes originate, more or less directly, as shown graphically

by the `pstree` program. `init` is the top of a tree structure whose branches spread further and further down.

```
wolfgang@meitner> pstree
init-+-acpid
     |-bonobo-activati
     |-cron
     |-cupsd
     |-2*[dbus-daemon]
     |-dbus-launch
     |-dcopserver
     |-dhcpcd
     |-esd
     |-eth1
     |-events/0
     |-gam_server
     |-gconfd-2
     |-gdm---gdm-+-X
     |           '-startkde-+-kwrapper
     |                      '-ssh-agent
     |-gnome-vfs-daemo
     |-gpg-agent
     |-hald-addon-acpi
     |-kaccess
     |-kded
     |-kdeinit-+-amarokapp---2*[amarokapp]
     |         |-evolution-alarm
     |         |-kinternet
     |         |-kio_file
     |         |-klauncher
     |         |-konqueror
     |         |-konsole---bash-+-pstree
     |         |                '-xemacs
     |         |-kwin
     |         |-nautilus
     |         '-netapplet
     |-kdesktop
     |-kgpg
     |-khelper
     |-kicker
     |-klogd
     |-kmix
     |-knotify
     |-kpowersave
     |-kscd
     |-ksmserver
     |-ksoftirqd/0
     |-kswapd0
     |-kthread-+-aio/0
     |         |-ata/0
     |         |-kacpid
     |         |-kblockd/0
     |         |-kgameportd
     |         |-khubd
```

```
        |         |-kseriod
        |         |-2*[pdflush]
        |         '-reiserfs/0
...
```

How this tree structure spreads is closely connected with how new processes are generated. For this purpose, UNIX uses two mechanisms called *fork* and *exec*.

1. **fork** — Generates an exact copy of the current process that differs from the parent process only in its PID (*process identification*). After the system call has been executed, there are two processes in the system, both performing the same actions. The memory contents of the initial process are duplicated — at least in the view of the program. Linux uses a well-known technique known as *copy on write* that allows it to make the operation much more efficient by deferring the copy operations until either parent or child writes to a page — read-only accessed can be satisfied from the same page for both.

   A possible scenario for using `fork` is, for example, when a user opens a second browser window. If the corresponding option is selected, the browser executes a `fork` to duplicate its code and then starts the appropriate actions to build a new window in the child process.

2. **exec** — Loads a new program into an existing content and then executes it. The memory pages reserved by the old program are flushed, and their contents are replaced with new data. The new program then starts executing.

Threads

Processes are not the only form of program execution supported by the kernel. In addition to *heavy-weight processes* — another name for classical UNIX processes — there are also *threads*, sometimes referred to as *light-weight processes*. They have also been around for some time, and essentially, a process may consist of several threads that all share the same data and resources but take different paths through the program code. The thread concept is fully integrated into many modern languages — Java, for instance. In simple terms, a process can be seen as an executing program, whereas a thread is a program function or routine running in parallel to the main program. This is useful, for example, when Web browsers need to load several images in parallel. Usually, the browser would have to execute several `fork` and `exec` calls to generate parallel instances; these would then be responsible for loading the images and making data received available to the main program using some kind of communication mechanisms. Threads make this situation easier to handle. The browser defines a routine to load images, and the routine is started as a thread with multiple strands (each with different arguments). Because the threads and the main program share the same address space, data received automatically reside in the main program. There is therefore no need for any communication effort whatsoever, except to prevent the threads from stepping onto their feet mutually by accessing identical memory locations, for instance. Figure 1-2 illustrates the difference between a program with and without threads.

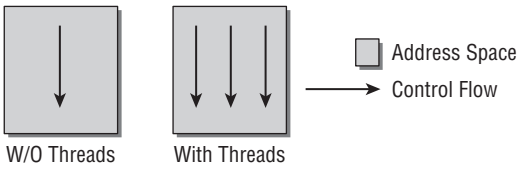

Figure 1-2: Processes with and without threads.

Linux provides the `clone` method to generate threads. This works in a similar way to `fork` but enables a precise check to be made of which resources are shared with the parent process and which are generated independently for the thread. This fine-grained distribution of resources extends the classical thread concept and allows for a more or less continuous transition between thread and processes.

Namespaces

During the development of kernel 2.6, support for namespaces was integrated into numerous subsystems. This allows different processes to have different views of the system. Traditionally, Linux (and UNIX in general) use numerous global quantities, for instance, process identifiers: Every process in the system is equipped with a unique identifier (ID), and this ID can be employed by users (or other processes) to refer to the process — by sending it a signal, for instance. With namespaces, formerly global resources are grouped differently: Every namespace can contain a specific set of PIDs, or can provide different views of the filesystem, where mounts in one namespace do not propagate into different namespaces.

Namespaces are useful; for example, they are beneficial for hosting providers: Instead of setting up one physical machine per customer, they can instead use *containers* implemented with namespaces to create multiple views of the system where each seems to be a complete Linux installation from within the container and does not interact with other containers: They are separated and segregated from each other. Every instance looks like a single machine running Linux, but in fact, many such instances can operate simultaneously on a physical machine. This helps use resources more effectively. In contrast to full virtualization solutions like KVM, only a single kernel needs to run on the machine and is responsible to manage all containers.

Not all parts of the kernel are yet fully aware of namespaces, and I will discuss to what extent support is available when we analyze the various subsystems.

1.3.3 Address Spaces and Privilege Levels

Before we start to discuss virtual address spaces, there are some notational conventions to fix. Throughout this book I use the abbreviations KiB, MiB, and GiB as units of size. The conventional units KB, MB, and GB are not really suitable in information technology because they represent decimal powers (10^3, 10^6, and 10^9) although the binary system is the basis ubiquitous in computing. Accordingly KiB stands for 2^{10}, MiB for 2^{20}, and GiB for 2^{30} bytes.

Because memory areas are addressed by means of pointers, the word length of the CPU determines the maximum size of the address space that can be managed. On 32-bit systems such as IA-32, PPC, and m68k, these are $2^{32} = 4$ GiB, whereas on more modern 64-bit processors such as Alpha, Sparc64, IA-64, and AMD64, 2^{64} bytes can be managed.

The maximal size of the address space is not related to how much physical RAM is actually available, and therefore it is known as the *virtual address space*. One more reason for this terminology is that every process in the system has the impression that it would solely live in this address space, and other processes are not present from their point of view. Applications do not need to care about other applications and can work as if they would run as the only process on the computer.

Linux divides virtual address space into two parts known as *kernel space* and *userspace* as illustrated in Figure 1-3.

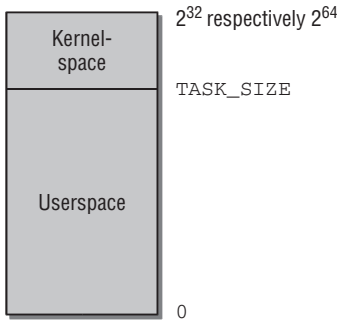

Figure 1-3: Division of virtual address space.

Every user process in the system has its own virtual address range that extends from 0 to TASK_SIZE. The area above (from TASK_SIZE to 2^{32} or 2^{64}) is reserved exclusively for the kernel — and may not be accessed by user processes. TASK_SIZE is an architecture-specific constant that divides the address space in a given ratio — in IA-32 systems, for instance, the address space is divided at 3 GiB so that the virtual address space for each process is 3 GiB; 1 GiB is available to the kernel because the total size of the virtual address space is 4 GiB. Although actual figures differ according to architecture, the general concepts do not. I therefore use these sample values in our further discussions.

This division *does not depend* on how much RAM is available. As a result of address space virtualization, *each* user process thinks it has 3 GiB of memory. The userspaces of the individual system processes are totally separate from each other. The kernel space at the top end of the virtual address space is always the same, regardless of the process currently executing.

Notice that the picture can be more complicated on 64-bit machines because these tend to use less than 64 bits to actually manage their huge principal virtual address space. Instead of 64 bits, they employ a smaller number, for instance, 42 or 47 bits. Because of this, the effectively addressable portion of the address space is smaller than the principal size. However, it is still larger than the amount of RAM that will ever be present in the machine, and is therefore completely sufficient. As an advantage, the CPU can save some effort because less bits are required to manage the effective address space than are required to address the complete virtual address space. The virtual address space will contain holes that are not addressable in principle in such cases, so the simple situation depicted in Figure 1-3 is not fully valid. We will come back to this topic in more detail in Chapter 4.

Privilege Levels

The kernel divides the virtual address space into two parts so that it is able to protect the individual system processes from each other. All modern CPUs offer several privilege levels in which processes can reside. There are various prohibitions in each level including, for example, execution of certain assembly language instructions or access to specific parts of virtual address space. The IA-32 architecture uses a system of four privilege levels that can be visualized as rings. The inner rings are able to access more functions, the outer rings less, as shown in Figure 1-4.

Whereas the Intel variant distinguishes four different levels, Linux uses only two different modes — kernel mode and user mode. The key difference between the two is that access to the memory area above TASK_SIZE — that is, kernel space — is forbidden in user mode. User processes are not able to manipulate or read the data in kernel space. Neither can they execute code stored there. This is the sole domain

of the kernel. This mechanism prevents processes from interfering with each other by unintentionally influencing each other's data.

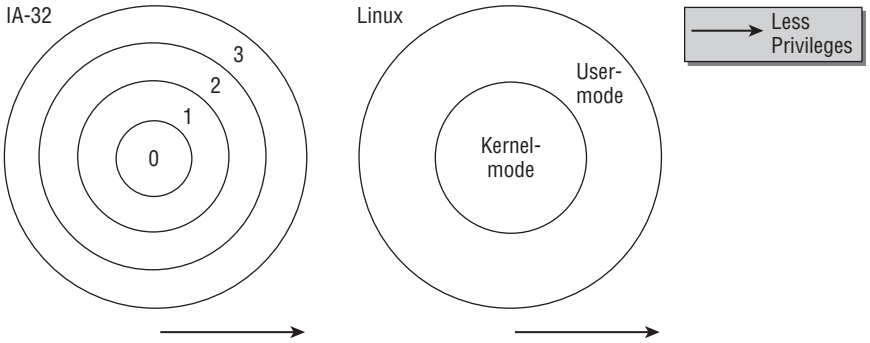

Figure 1-4: Ring system of privilege levels.

The switch from user to kernel mode is made by means of special transitions known as *system calls*; these are executed differently depending on the system. If a normal process wants to carry out any kind of action affecting the entire system (e.g., manipulating I/O devices), it can do this only by issuing a request to the kernel with the help of a system call. The kernel first checks whether the process is *permitted* to perform the desired action and then performs the action on its behalf. A return is then made to user mode.

Besides executing code on behalf of a user program, the kernel can also be activated by asynchronous hardware interrupts, and is then said to run in *interrupt context*. The main difference to running in process context is that the userspace portion of the virtual address space must not be accessed. Because interrupts occur at random times, a random userland process is active when an interrupt occurs, and since the interrupt will most likely be unconnected with the cause of the interrupt, the kernel has no business with the contents of the current userspace. When operating in interrupt context, the kernel must be more cautious than normal; for instance, it must not go to sleep. This requires extra care when writing interrupt handlers and is discussed in detail in Chapter 2. An overview of the different execution contexts is given in Figure 1-5.

Besides normal processes, there can also be *kernel threads* running on the system. Kernel threads are also not associated with any particular userspace process, so they also have no business dealing with the user portion of the address space. In many other respects, kernel threads behave much more like regular userland applications, though: In contrast to a kernel operating in interrupt context, they may go to sleep, and they are also tracked by the scheduler like every regular process in the system. The kernel uses them for various purposes that range from data synchronization of RAM and block devices to helping the scheduler distribute processes among CPUs, and we will frequently encounter them in the course of this book.

Notice that kernel threads can be easily identified in the output of `ps` because their names are placed inside brackets:

```
wolfgang@meitner> ps fax
  PID TTY      STAT   TIME COMMAND
    2 ?        S<     0:00 [kthreadd]
    3 ?        S<     0:00  _ [migration/0]
    4 ?        S<     0:00  _ [ksoftirqd/0]
```

```
    5 ?        S<     0:00  _ [migration/1]
    6 ?        S<     0:00  _ [ksoftirqd/1]
    7 ?        S<     0:00  _ [migration/2]
    8 ?        S<     0:00  _ [ksoftirqd/2]
    9 ?        S<     0:00  _ [migration/3]
   10 ?        S<     0:00  _ [ksoftirqd/3]
   11 ?        S<     0:00  _ [events/0]
   12 ?        S<     0:00  _ [events/1]
   13 ?        S<     0:00  _ [events/2]
   14 ?        S<     0:00  _ [events/3]
   15 ?        S<     0:00  _ [khelper]
...
15162 ?        S<     0:00  _ [jfsCommit]
15163 ?        S<     0:00  _ [jfsSync]
```

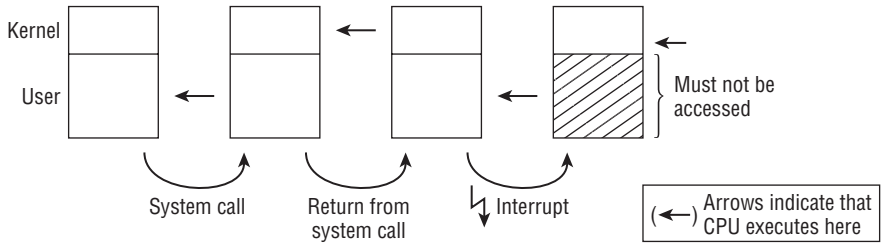

Figure 1-5: Execution in kernel and user mode. Most of the time, the CPU executes code in userspace. When the application performs a system call, a switch to kernel mode is employed, and the kernel fulfills the request. During this, it may access the user portion of the virtual address space. After the system call completes, the CPU switches back to user mode. A hardware interrupt also triggers a switch to kernel mode, but this time, the userspace portion must not be accessed by the kernel.

On multiprocessor systems, many threads are started on a per-CPU basis and are restricted to run on only one specific processor. This is represented by a slash and the number of the CPU that are appended to the name of the kernel thread.

Virtual and Physical Address Spaces

In most cases, a single virtual address space is bigger than the physical RAM available to the system. And the situation does not improve when *each* process has its own virtual address space. The kernel and CPU must therefore consider how the physical memory actually available can be mapped onto virtual address areas.

The preferred method is to use page tables to allocate *virtual* addresses to *physical* addresses. Whereas virtual addresses relate to the combined user and kernel space of a process, physical addresses are used to address the RAM actually available. This principle is illustrated in Figure 1-6.

The virtual address spaces of both processes shown in the figure are divided into portions of equal size by the kernel. These portions are known as *pages*. Physical memory is also divided into pages of the same size.

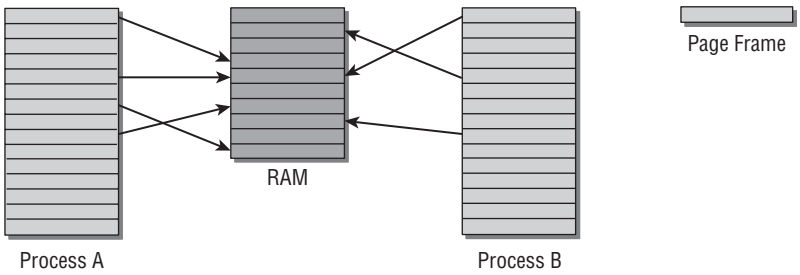

Figure 1-6: Virtual and physical addresses.

The arrows in Figure 1-6 indicate how the pages in the virtual address spaces are distributed across the physical pages. For example, virtual page 1 of process A is mapped to physical page 4, while virtual page 1 of process B is mapped to the fifth physical page. This shows that virtual addresses change their meaning from process to process.

Physical pages are often called *page frames*. In contrast, the term *page* is reserved for pages in virtual address space.

Mapping between virtual address spaces and physical memory also enables the otherwise strict separation between processes to be lifted. Our example includes a page frame explicitly shared by both processes. Page 5 of A and page 1 of B both point to the physical page frame 5. This is possible because entries in both virtual address spaces (albeit at different positions) point to the same page. Since the kernel is responsible for mapping virtual address space to physical address space, it is able to decide which memory areas are to be shared between processes and which are not.

The figure also shows that not all pages of the virtual address spaces are linked with a page frame. This may be because either the pages are not used or because data have not been loaded into memory because they are not yet needed. It may also be that the page has been swapped out onto hard disk and will be swapped back in when needed.

Finally, notice that there are two equivalent terms to address the applications that run on behalf of the user. One of them is *userland*, and this is the nomenclature typically preferred by the BSD community for all things that do not belong to the kernel. The alternative is to say that an application runs in *userspace*. It should be noted that the term *userland* will always mean applications as such, whereas the term *userspace* can additionally not only denote applications, but also the portion of the virtual address space in which they are executed, in contrast to *kernel space*.

1.3.4 Page Tables

Data structures known as *page tables* are used to map virtual address space to physical address space. The easiest way of implementing the association between both would be to use an array containing an entry for each page in virtual address space. This entry would point to the associated page frame. But there is a problem. IA-32 architecture uses, for example, 4 KiB pages — given a virtual address space of 4 GiB, this would produce an array with a million entries. On 64-bit architectures, the situation is much worse. Because each process needs its own page tables, this approach is impractical because the entire RAM of the system would be needed to hold the page tables.

As most areas of virtual address spaces are not used and are therefore not associated with page frames, a far less memory-intensive model that fulfills the same purpose can be used: multilevel paging.

To reduce the size of page tables and to allow unneeded areas to be ignored, the architectures split each virtual address into multiple parts, as shown in Figure 1-7 (the bit positions at which the address is split differ according to architecture, but this is of no relevance here). In the example, I use a split of the virtual address into four components, and this leads to a *three*-level page table. This is what most architectures offer. However, some employ four-level page tables, and Linux also adopts four levels of indirection. To simplify the picture, I stick to a three-level variant here.

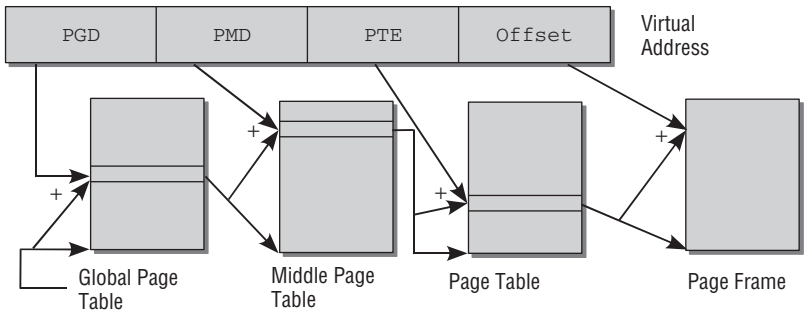

Figure 1-7: Splitting a virtual address.

The first part of the virtual address is referred to as a *page global directory* or PGD. It is used as an index in an array that exists exactly once for each process. Its entries are pointers to the start of further arrays called *page middle directories* or PMD.

Once the corresponding array has been found by reference to the PGD and its contents, the PMD is used as an index for the array. The page middle directory likewise consists of pointers to further arrays known as *page tables* or *page directories*.

The *PTE* (or *page table entry*) part of the virtual address is used as an index to the page table. Mapping between virtual pages and page frames is achieved because the page table entries point to page frames.

The last part of the virtual address is known as an *offset*. It is used to specify a byte position within the page; after all, each address points to a uniquely defined byte in address space.

A particular feature of page tables is that no page middle tables or page tables need be created for areas of virtual address space that are not needed. This saves a great deal of RAM as compared to the single-array method.

Of course, this method also has a downside. Each time memory is accessed, it is necessary to run through the entire chain to obtain the physical address from the virtual address. CPUs try to speed up this process in two ways:

1. A special part of the CPU known as a *memory management unit* (*MMU*) is optimized to perform referencing operations.

2. The addresses that occur most frequently in address translation are held in a fast CPU cache called a *Translation Lookaside Buffer* (*TLB*). Translation is accelerated because the address data in the cache are immediately available without needing to access the page tables and therefore the RAM.

   While caches are operated transparently on many architectures, some require special attention from the kernel, which especially implies that their contents must be invalidated whenever the contents of the page tables have been changed. Corresponding calls must be present in every part of the kernel that manipulates page tables. If the kernel is compiled for an architecture that does not require such operations, it automatically ensures that the calls are represented by do-nothing operations.

Interaction with the CPU

The IA-32 architecture uses a two-level-only method to map virtual addresses to physical addresses. The size of the address space in 64-bit architectures (Alpha, Sparc64, IA-64, etc.) mandates a three-level or four-level method, and the architecture-*independent* part of the kernel always assumes a four-level page table.

The architecture-*dependent* code of the kernel for two- and three-level CPUs must therefore emulate the missing levels by dummy page tables. Consequently, the remaining memory management code can be implemented independently of the CPU used.

Memory Mappings

Memory mappings are an important means of abstraction. They are used at many points in the kernel and are also available to user applications. Mapping is the method by which data from an arbitrary source are transferred into the virtual address space of a process. The address space areas in which mapping takes place can be processed using normal methods in the same way as regular memory. However, any changes made are transferred automatically to the original data source. This makes it possible to use identical functions to process totally different things. For example, the contents of a file can be mapped into memory. A process then need only read the contents of memory to access the contents of the file, or write changes to memory in order to modify the contents of the file. The kernel automatically ensures that any changes made are implemented in the file.

Mappings are also used directly in the kernel when implementing device drivers. The input and output areas of peripheral devices can be mapped into virtual address space; reads and writes to these areas are then redirected to the devices by the system, thus greatly simplifying driver implementation.

1.3.5 Allocation of Physical Memory

When it allocates RAM, the kernel must keep track of which pages have already been allocated and which are still free in order to prevent two processes from using the same areas in RAM. Because memory allocation and release are very frequent tasks, the kernel must also ensure that they are completed as quickly as possible. The kernel can allocate only whole page frames. Dividing memory into smaller portions is delegated to the standard library in userspace. This library splits the page frames received from the kernel into smaller areas and allocates memory to the processes.

The Buddy System

Numerous allocation requests in the kernel must be fulfilled by a continuous range of pages. To quickly detect where in memory such ranges are still available, the kernel employs an old, but proven technique: The *buddy system*.

Free memory blocks in the system are always grouped as two buddies. The buddies can be allocated independently of each other; if, however, both remain unused at the same time, the kernel merges them into a larger pair that serves as a buddy on the next level. Figure 1-8 demonstrates this using an example of a buddy pair consisting initially of two blocks of 8 pages.

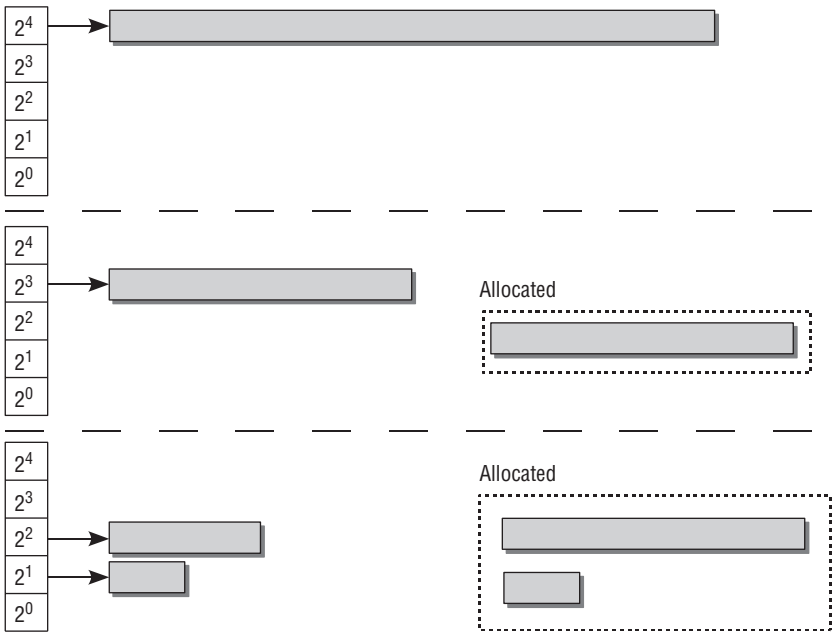

Figure 1-8: The buddy system.

All buddies of the same size (1, 2, 4, 8, 16, ... pages) are managed by the kernel in a special list. The buddy pair with two times 8 (16) pages is also in this list.

If the system now requires 8 page frames, it splits the block consisting of 16 page frames into two buddies. While one of the blocks is passed to the application that requested memory, the remaining 8 page frames are placed in the list for 8-page memory blocks.

If the next request requires only 2 contiguous page frames, the block consisting of 8 blocks is split into 2 buddies, each comprising 4 page frames. One of the blocks is put back into the buddy lists, while the other is again split into 2 buddies consisting of 2 blocks of two pages. One is returned to the buddy system, while the other is passed to the application.

When memory is returned by the application, the kernel can easily see by reference to the addresses whether a buddy pair is reunited and can then merge it into a larger unit that is put back into the buddy list — exactly the reverse of the splitting process. This increases the likelihood that larger memory blocks are available.

When systems run for longer periods — it is not unusual for servers to run for several weeks or even months, and many desktop systems also tend to reach long uptime — a memory management problem known as *fragmentation* occurs. The frequent allocation and release of page frames may lead to a situation in which several page frames are free in the system but they are scattered throughout physical address space — in other words, there are no larger *contiguous* blocks of page frames, as would be desirable for performance reasons. This effect is reduced to some extent by the buddy system but not completely eliminated. Single reserved pages that sit in the middle of an otherwise large continuous free range can eliminate coalescing of this range very effectively. During the development of kernel 2.6.24, some effective measures were added to prevent memory fragmentation, and I discuss the underlying mechanisms in more detail in Chapter 3.

The Slab Cache

Often the kernel itself needs memory blocks much smaller than a whole page frame. Because it cannot use the functions of the standard library, it must define its own, additional layer of memory management that builds on the buddy system and divides the pages supplied by the buddy system into smaller portions. The method used not only performs allocation but also implements a generic cache for frequently used small objects; this cache is known as a *slab cache*. It can be used to allocate memory in two ways:

1. For frequently used objects, the kernel defines its own cache that contains only instances of the desired type. Each time one of the objects is required, it can be quickly removed from the cache (and returned there after use); the slab cache automatically takes care of interaction with the buddy system and requests new page frames when the existing caches are full.
2. For the general allocation of smaller memory blocks, the kernel defines a set of slab caches for various object sizes that it can access using the same functions with which we are familiar from userspace programming; a prefixed `k` indicates that these functions are associated with the kernel: `kmalloc` and `kfree`.

While the slab allocator provides good performance across a wide range of workloads, some scalability problems with it have arisen on really large supercomputers. On the other hand of the scale, the overhead of the slab allocator may be too much for really tiny embedded systems. The kernel comes with two drop-in replacements for the slab allocator that provide better performance in these use cases, but offer the same interface to the rest of the kernel such that it need not be concerned with which low-level allocator is actually compiled in. Since slab allocation is still the standard methods of the kernel, I will, however, not discuss these alternatives in detail. Figure 1-9 summarizes the connections between buddy system, slab allocator, and the rest of the kernel.

Swapping and Page Reclaim

Swapping enables available RAM to be enlarged virtually by using disk space as extended memory. Infrequently used pages can be written to hard disk when the kernel requires more RAM. Once the data

are actually needed, the kernel swaps them back into memory. The concept of *page faults* is used to make this operation transparent to applications. Swapped-out pages are identified by a special entry in the page table. When a process attempts to access a page of this kind, the CPU initiates a page fault that is intercepted by the kernel. The kernel then has the opportunity to swap the data on disk into RAM. The user process then resumes. Because it is unaware of the page fault, swapping in and out of the page is totally invisible to the process.

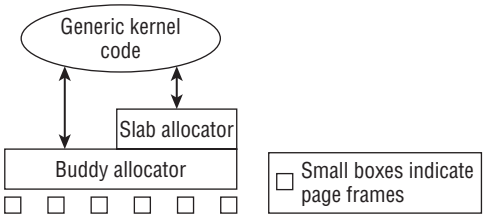

Figure 1-9: Page frame allocation is performed by the buddy system, while the slab allocator is responsible for small-sized allocations and generic kernel caches.

Page reclaim is used to synchronize modified mappings with underlying block devices — for this reason, it is sometimes referred to simply as *writing back data*. Once data have been flushed, the page frame can be used by the kernel for other purposes (as with swapping). After all, the kernel data structures contain all the information needed to find the corresponding data on the hard disk when they are again required.

1.3.6 Timing

The kernel must be capable of measuring time and time differences at various points — when scheduling processes, for example. *Jiffies* are one possible time base. A global variable named `jiffies_64` and its 32-bit counterpart `jiffies` are incremented periodically at constant time intervals. The various timer mechanisms of the underlying architectures are used to perform these updates — each computer architecture provides some means of executing periodic actions, usually in the form of timer interrupts.

Depending on architecture, `jiffies` is incremented with a frequency determined by the central constant `HZ` of the kernel. This is usually on the range between 1,000 and 100; in other words, the value of `jiffies` is incremented between 1,000 and 100 times per second.

Timing based on `jiffies` is relatively coarse-grained because 1,000 Hz is not an excessively large frequency nowadays. With *high-resolution timers*, the kernel provides additional means that allows for keeping time in the regime of nanosecond precision and resolution, depending on the capabilities of the underlying hardware.

It is possible to make the periodic tick *dynamic*. When there is little to do and no need for frequent periodic actions, it does not make sense to periodically generate timer interrupts that prevent the processor from powering down into deep sleep states. This is helpful in systems where power is scarce, for instance, laptops and embedded systems.

1.3.7 System Calls

System calls are the classical method of enabling user processes to interact with the kernel. The POSIX standard defines a number of system calls and their effect as implemented on all POSIX-compliant systems including Linux. Traditional system calls are grouped into various categories:

- **Process Management** — Creating new tasks, querying information, debugging
- **Signals** — Sending signals, timers, handling mechanisms
- **Files** — Creating, opening, and closing files, reading from and writing to files, querying information and status
- **Directories and Filesystem** — Creating, deleting, and renaming directories, querying information, links, changing directories
- **Protection Mechanisms** — Reading and changing UIDs/GIDs, and namespace handling
- **Timer Functions** — Timer functions and statistical information

Demands are placed on the kernel in all these functions. They cannot be implemented in a normal user library because special protection mechanisms are needed to ensure that system stability and/or security are not endangered. In addition, many calls are reliant on kernel-internal structures or functions to yield desired data or results — this also dictates against implementation in userspace. When a system call is issued, the processor must change the privilege level and switch from user mode to system mode. There is no standardized way of doing this in Linux as each hardware platform offers specific mechanisms. In some cases, different approaches are implemented on the same architecture but depend on processor type. Whereas Linux uses a special software interrupt to execute system calls on IA-32 processors, the software emulation (iBCS emulator) of other UNIX systems on IA-32 employs a different method to execute binary programs (for assembly language aficionados: the `lcall7` or `lcall27` gate). Modern variants of IA-32 also have their own assembly language statement for executing system calls; this was not available on old systems and cannot therefore be used on all machines. What all variants have in common is that system calls are the only way of enabling user processes to switch in their own incentive from user mode to kernel mode in order to delegate system-critical tasks.

1.3.8 Device Drivers, Block and Character Devices

The role of device drivers is to communicate with I/O devices attached to the system; for example, hard disks, floppies, interfaces, sound cards, and so on. In accordance with the classical UNIX maxim that ''*everything is a file*,'' access is performed using device files that usually reside in the `/dev` directory and can be processed by programs in the same way as regular files. The task of a device driver is to support application communication via device files; in other words, to enable data to be read from and written to a device in a suitable way.

Peripheral devices belong to one of the following two groups:

1. **Character Devices** — Deliver a continuous stream of data that applications read sequentially; generally, random access is not possible. Instead, such devices allow data to be read and written byte-by-byte or character-by-character. Modems are classical examples of character devices.

2. **Block Devices** — Allow applications to address their data randomly and to freely select the position at which they want to read data. Typical block devices are hard disks because applications can address any position on the disk from which to read data. Also, data can be read or written only in multiples of block units (usually 512 bytes); character-based addressing, as in character devices, is not possible.

   Programming drivers for block devices is much more complicated than for character devices because extensive caching mechanisms are used to boost system performance.

1.3.9 Networks

Network cards are also controlled by device drivers but assume a special status in the kernel because they cannot be addressed using device files. This is because data are packed into various protocol layers during network communication. When data are received, the layers must be disassembled and analyzed by the kernel before the payload data are passed to the application. When data are sent, the kernel must first pack the data into the various protocol layers prior to dispatch.

However, to support work with network connections via the file interface (in the view of applications), Linux uses *sockets* from the BSD world; these act as agents between the application, file interface, and network implementation of the kernel.

1.3.10 Filesystems

Linux systems are made up of many thousands or even millions of files whose data are stored on hard disks or other block devices (e.g., ZIP drives, floppies, CD-ROMs, etc.). Hierarchical filesystems are used; these allow stored data to be organized into directory structures and also have the job of linking other meta-information (owners, access rights, etc.) with the actual data. Many different filesystem approaches are supported by Linux — the standard filesystems Ext2 and Ext3, ReiserFS, XFS, VFAT (for reasons of compatibility with DOS), and countless more. The concepts on which they build differ drastically in part. Ext2 is based on inodes, that is, it makes a separate management structure known as an *inode* available on disk for each file. The inode contains not only all meta-information but also pointers to the associated data blocks. Hierarchical structures are set up by representing directories as regular files whose data section includes pointers to the inodes of all files contained in the directory. In contrast, ReiserFS makes extensive use of tree structures to deliver the same functionality.

The kernel must provide an additional software layer to abstract the special features of the various low-level filesystems from the application layer (and also from the kernel itself). This layer is referred to as the VFS (*virtual filesystem* or *virtual filesystem switch*). It acts as an interface downward (this interface must be implemented by all filesystems) and upward (for system calls via which user processes are ultimately able to access filesystem functions). This is illustrated in Figure 1-10.

1.3.11 Modules and Hotplugging

Modules are used to dynamically add functionality to the kernel at run time — device drivers, filesystems, network protocols, practically any subsystem[3] of the kernel can be modularized. This removes one of the significant disadvantages of monolithic kernels as compared with microkernel variants.

[3]With the exception of basic functions, such as memory management, which are always needed.

Modules can also be unloaded from the kernel at run time, a useful aspect when developing new kernel components.

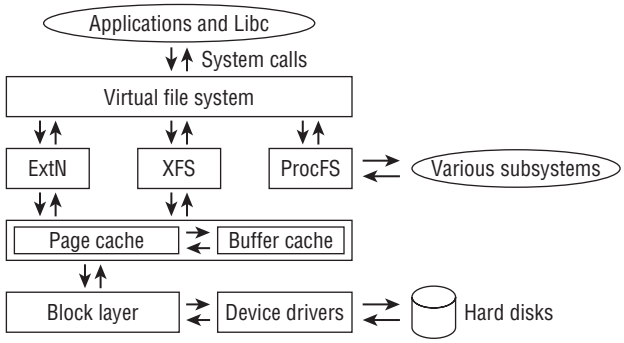

Figure 1-10: Overview of how the virtual filesystem layer, filesystem implementations, and the block layer interoperate.

Basically, modules are simply normal programs that execute in kernel space rather than in userspace. They must also provide certain sections that are executed when the module is initialized (and terminated) in order to register and de-register the module functions with the kernel. Otherwise, module code has the same rights (and obligations) as normal kernel code and can access all the same functions and data as code that is permanently compiled into the kernel.

Modules are an essential requisite to support for *hotplugging*. Some buses (e.g., USB and FireWire) allow devices to be connected while the system is running without requiring a system reboot. When the system detects a new device, the requisite driver can be automatically added to the kernel by loading the corresponding module.

Modules also enable kernels to be built to support all kinds of devices that the kernel can address without unnecessarily bloating kernel size. Once attached hardware has been detected, only the requisite modules are loaded, and the kernel remains free of superfluous drivers.

A long-standing issue in the kernel community revolves around the support of binary-only modules, that is, modules for which no source code is available. While binary-only modules are omnipresent on most proprietary operating systems, many kernel developers see them (at least!) as an incarnation of the devil: Since the kernel is developed as open-source software, they believe that modules should also be published as open source, for a variety of both legal and technical reasons. There are, indeed, strong arguments to support this reasoning (and besides, I also follow these), but they are not shared by some commercial companies that tend to think that opening up their driver sources would weaken their business position.

It is currently possible to load binary-only modules into the kernel, although numerous restrictions apply for them. Most importantly, they may not access any functions that are explicitly only made available to GPL-licensed code. Loading a binary-only module *taints* the kernel, and whenever something bad occurs, the fault is naturally attributed to the tainting module. If a kernel is tainted, this will be marked on crash dumps, for instance, and kernel developers will be very unsupportive in solving the issue that led to the crash — since the binary module could have given every part of the kernel a good shaking, it cannot

be assumed that the kernel still works as intended, and support is better left to the manufacturer of the offending module.

Loading binary-only modules is not the only possibility for tainting a kernel. This happens also when, for instance, the machine has experienced certain bad exceptions, when a SMP system is built with CPUs that do not officially support multiprocessing by their specification, and other similar reasons.

1.3.12 Caching

The kernel uses *caches* to improve system performance. Data read from slow block devices are held in RAM for a while, even if they are no longer needed at the time. When an application next accesses the data, they can be read from fast RAM, thus bypassing the slow block device. Because the kernel implements access to block devices by means of page memory mappings, caches are also organized into pages, that is, whole pages are cached, thus giving rise to the name *page cache*.

The far less important *buffer cache* is used to cache data that are not organized into pages. On traditional UNIX systems, the buffer cache serves as the main system cache, and the same approach was used by Linux a long, long time ago. By now, the buffer cache has mostly been superseded by the page cache.

1.3.13 List Handling

A recurring task in C programs is the handling of doubly linked lists. The kernel too is required to handle such lists. Consequently, I will make frequent mention of the standard list implementation of the kernel in the following chapters. At this point, I give a brief introduction to the list handling API.

Standard lists as provided by the kernel can be used to link data structures of any type with each other. It is explicitly *not* type-safe. The data structures to be listed must contain an element of the `list_head` type; this accommodates the forward and back pointers. If a data structure is to be organized in several lists — and this is not unusual — several `list_head` elements are needed.

<list.h>

```
struct list_head {
        struct list_head *next, *prev;
};
```

This element could be placed in a data structure as follows:

```
struct task_struct {
...
   struct list_head run_list;
...
};
```

The starting point for linked lists is again an instance of `list_head` that is usually declared and initialized by the `LIST_HEAD(list_name)` macro. In this way, the kernel produces a cyclic list, as shown in Figure 1-11. It permits access to the first and last element of a list in $\mathbb{O}(1)$, that is, in always the same, constant time regardless of the list size.

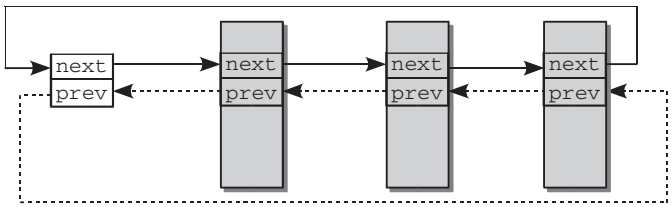

Figure 1-11: Doubly linked standard list.

`struct list_head` is called a *list element* when it is held in a data structure. An element that serves as the starting point for a list is called a *list head*.

Pointers that connect head and tail elements of a list tend to clutter up images and often obstruct the principal intention of a figure, namely, to briefly summarize the connections of various kernel data structures. I thus usually *omit* the connection between list head and list tail in figures. The above list is in the remainder of this book therefore represented as shown in Figure 1-12. This allows for concentrating on the essential details without having to waste space for irrelevant list pointers.

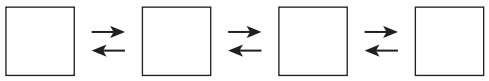

Figure 1-12: Simplified illustration of a doubly linked list. Notice that the connection between list head and list tail is *not* displayed, although it is present in kernel memory.

There are several standard functions for handling and processing lists. We will come across them again and again in the following chapters (the data type of their arguments is `struct list_head`).

- `list_add(new, head)` inserts `new` right after the existing `head` element.
- `list_add_tail(new, head)` inserts `new` right before the element specified by `head`. If the list head is specified for `head`, this causes the new element to be inserted at the end of the list because of the cyclic nature of the list (this gives the function its name).
- `list_del(entry)` deletes an entry from a list.
- `list_empty(head)` checks if a list is empty, that is, if it does not contain any elements.
- `list_splice(list, head)` combines two lists by inserting the list in `list` after the `head` element of an existing list.
- `list_entry` must be used to find a list element; at first glance, its call syntax appears to be quite complicated: `list_entry(ptr, type, member)`. `ptr` is a pointer to the `list_head` instance of the

data structure, `type` is its type, and `member` is the element name used for the list element. The following sample call would be needed to find a `task_struct` instance of a list:

```
struct task_struct = list_entry(ptr, struct task_struct, run_list)
```

Explicit type specification is required because list implementation is *not* type-safe. The list element must be specified to find the correct element if there are data structures that are included in several lists.[4]

- `list_for_each(pos, head)` must be used to iterate through all elements of a list. `pos` indicates the current position in the list, while `head` specifies the list head.

```
struct list_head *p;

list_for_each(p, &list)
        if (condition)
                return list_entry(p, struct task_struct, run_list);
return NULL;
```

1.3.14 Object Management and Reference Counting

All over the kernel, the need to keep track of instances of C structures arises. Despite the fact that these objects will be used in the most varying forms, some operations are very similar across subsystems — just consider reference counting. This leads to code duplication. Since this is a bad thing, the kernel has adopted generic methods to manage kernel objects during the development of 2.5. The framework is, however, not just required to prevent code duplication. It also allows for providing a coherent view on objects managed by different parts of the kernel, and this information can be brought to good use in many parts of the kernel, for instance, for power management.

The generic kernel object mechanism can be used to perform the following operations on objects:

- Reference counting
- Management of lists (sets) of objects
- Locking of sets
- Exporting object properties into userspace (via the `sysfs` filesystem)

Generic Kernel Objects

The following data structure that is embedded in other data structures is used as a basis.

<kobject.h>

```
struct kobject {
        const char              * k_name;
        struct kref             kref;
        struct list_head        entry;
        struct kobject          * parent;
        struct kset             * kset;
        struct kobj_type        * ktype;
        struct sysfs_dirent     * sd;
};
```

[4] Even if there is only one list element in the structure, this entry is used to find the correct start address of the instance by means of pointer arithmetic; the address is translated into the required data type by means of type conversion. I deal with this in more detail in the appendix on C programming.

It is essential that `kobjects` are not linked with other data structures by means of pointers but are directly embedded. Managing the kernel object itself amounts to managing the whole containing object this way. Since `struct kobject` is embedded into many data structures of the kernel, the developers take care to keep it small. Adding a single new element to this data structure results in a size increase of many other data structures. Embedded kernel objects look as follows:

```
struct sample {
...
    struct kobject kobj;
...
};
```

The meanings of the individual elements of `struct kobject` are as follows:

- `k_name` is a text name exported to userspace using `sysfs`. Sysfs is a virtual filesystem that allows for exporting various properties of the system into userspace. Likewise `sd` supports this connection, and I will come back to this in Chapter 10.
- `kref` holds the general type `struct kref` designed to simplify reference management. I discuss this below.
- `entry` is a standard list element used to group several `kobject`s in a list (known as a set in this case).
- `kset` is required when an object is grouped with other objects in a set.
- `parent` is a pointer to the parent element and enables a hierarchical structure to be established between `kobject`s.
- `ktype` provides more detailed information on the data structure in which a `kobject` is embedded. Of greatest importance is the destructor function that returns the resources of the embedding data structure.

The similarity between the name `kobject` and the *object* concept of, well, object-oriented languages like C++ or Java is by no means coincidental: The `kobject` abstraction indeed allows for using object-oriented techniques in the kernel, but without requiring all the extra mechanics (and bloat, and overhead) of C++.

Table 1-1 lists the standard operations provided by the kernel to manipulate `kobject` instances, and therefore effectively act on the embedding structure.

The layout of the `kref` structure used to manage references is as follows:

<kref.h>

```
struct kref {
        atomic_t refcount;
};
```

`refcount` is an atomic data type to specify the number of positions in the kernel at which an object is currently being used. When the counter reaches 0, the object is no longer needed and can therefore be removed from memory.

Table 1-1: Standard Methods for Processing `kobject`s

Function	Meaning
`kobject_get`, `kobject_put`	Increments or decrements the reference counter of a `kobject`
`kobject_(un)register`	Registers or removes `obj` from a hierarchy (the object is added to the existing set (if any) of the parent element; a corresponding entry is created in the `sysfs` filesystem).
`kobject_init`	Initializes a `kobject`; that is, it sets the reference counter to its initial value and initializes the list elements of the object.
`kobect_add`	Initializes a kernel object and makes it visible in sysfs
`kobject_cleanup`	Releases the allocated resources when a `kobject` (and therefore the embedding object) is no longer needed

Encapsulation of the single value in a structure was chosen to prevent direct manipulation of the value. `kref_init` must always be used for initialization. If an object is in use, `kref_get` must be invoked beforehand to increment the reference counter. `kref_put` decrements the counter when the object is no longer used.

Sets of Objects

In many cases, it is necessary to group different kernel objects into a set — for instance, the set of all character devices or the set of all PCI-based devices. The data structure provided for this purpose is defined as follows:

<kobject.h>

```
struct kset {
        struct kobj_type        * ktype;
        struct list_head        list;
...
        struct kobject          kobj;
        struct kset_uevent_ops  * uevent_ops;
};
```

Interestingly, the `kset` serves as the first example for the use of kernel objects. Since the management structure for sets is nothing other than a kernel object, it can be managed via the previously discussed `struct kobj`. Indeed, an instance is embedded via `kobj`. It has nothing to do with the `kobject`s collected in the set, but only serves to manage the properties of the `kset` object itself.

The other members have the following meaning:

- `ktype` points to a further object that generalizes the behavior of the `kset`.
- `list` is used to build a list of all kernel objects that are a member of the set.
- `uevent_ops` provides several function pointers to methods that relay information about the state of the set to userland. This mechanism is used by the core of the driver model, for instance, to format messages that inform about the addition of new devices.

Another structure is provided to group common features of kernel objects. It is defined as follows:

```
<kobject.h>
struct kobj_type {
...
        struct sysfs_ops        * sysfs_ops;
        struct attribute        ** default_attrs;
};
```

Note that a `kobj_type` is *not* used to collect various kernel objects — this is already managed by `ksets`. Instead, it provides an interface to the sysfs filesystem (discussed in Section 10.3). If multiple objects export similar information via the filesystem, then this can be simplified by using a single ktype to provide the required methods.

Reference Counting

Reference counting is used to detect from how many places in the kernel an object is used. Whenever one part of the kernel needs information contained in one object, it increments the reference count, and when it does not need the information anymore, the count is decremented. Once the count has dropped to 0, the kernel knows that the object is not required anymore, and that it is safe to release it from memory. The kernel provides the following data structure to handle reference counting:

```
<kref.h>
struct kref {
        atomic_t refcount;
};
```

The data structure is really simple in that it only provides a generic, atomic reference count. ''Atomic'' means in this context that incrementing and decrementing the variable is also safe on multiprocessor systems, where more than one code path can access an object at the same time. Chapter 5 discusses the need for this in more detail.

The auxiliary methods `kref_init`, `kref_get`, and `kref_put` are provided to initialize, increment, or decrement the reference counter. This might seem trivial at a first glance. Nevertheless, it helps to avoid excessive code duplication because such reference counts together with the aforementioned operations are used all over the kernel.

> **Although manipulating the reference counter this way is safe against concurrency issues, this does *not* imply that the surrounding data structure is safe against concurrent access! Kernel code needs to employ further means to ensure that access to data structures does not cause any problems when this can happen from multiple processors simultaneously, and I discuss these issues in Chapter 5.**

Finally, notice that the kernel contains some documentation related to kernel objects in `Documentation/kobject.txt`.

1.3.15 Data Types

Some issues related to data types are handled differently in the kernel in comparison to userland programs.

Type Definitions

The kernel uses `typedef` to define various data types in order to make itself independent of architecture-specific features because of the different bit lengths for standard data types on individual processors. The definitions have names such as `sector_t` (to specify a sector number on a block device), `pid_t` (to indicate a process identifier), and so on, and are defined by the kernel in architecture-specific code in such a way as to ensure that they represent the applicable value range. Because it is not usually important to know on which fundamental data types the definitions are based, and for simplicity's sake, I do not always discuss the exact definitions of data types in the following chapters. Instead, I use them without further explanation — after all, they are simply non-compound standard data types under a different name.

> **`typedef`'d variables must not be accessed directly, but only via auxiliary functions that I introduce when we encounter the type. This ensures that they are properly manipulated, although the type definition is transparent to the user.**

At certain points, the kernel must make use of variables with an exact, clearly defined number of bits — for example, when data structures need to be stored on hard disk. To allow data to be exchanged between various systems (e.g., on USB sticks), the same external format must always be used, regardless of how data are represented internally in the computer.

To this end, the kernel defines several integer data types that not only indicate explicitly whether they are signed or unsigned, but also specify the *exact* number of bits they comprise. `__s8` and `__u8` are, for example, 8-bit integers that are either signed (`__s8`) or unsigned (`__u8`). `__u16` and `__s16`, `__u32` and `__s32`, and `__u64` and `__s64` are defined in the same way.

Byte Order

To represent numbers, modern computers use either the *big endian* or *little endian* format. The format indicates how multibyte data types are stored. With big endian ordering, the most significant byte is stored at the lowest address and the significance of the bytes decreases as the addresses increase. With little endian ordering, the least significant byte is stored at the lowest address and the significance of the bytes increases as the addresses increase (some architectures such as MIPS support both variants). Figure 1-13 illustrates the issue.

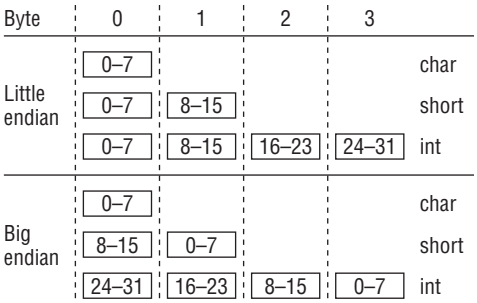

Figure 1-13: Composition of elementary data types depending on the endianness of the underlying architecture.

The kernel provides various functions and macros to convert between the format used by the CPU and specific representations: `cpu_to_le64` converts a 64-bit data type to little endian format, and `le64_to_cpu` does the reverse (if the architecture works with little endian format, the routines are, of course, no-ops; otherwise, the byte positions must be exchanged accordingly). Conversion routines are available for all combinations of 64, 32, and 16 bits for big and little endian.

Per-CPU Variables

A particularity that does not occur in normal userspace programming is per-CPU variables. They are declared with `DEFINE_PER_CPU(name, type)`, where `name` is the variable name and `type` is the data type (e.g., `int[3]`, `struct hash`, etc.). On single-processor systems, this is not different from regular variable declaration. On SMP systems with several CPUs, an instance of the variable is created for each CPU. The instance for a particular CPU is selected with `get_cpu(name, cpu)`, where `smp_processor_id()`, which returns the identifier of the active processor, is usually used as the argument for `cpu`.

Employing per-CPU variables has the advantage that the data required are more likely to be present in the cache of a processor and can therefore be accessed faster. This concept also skirts round several communication problems that would arise when using variables that can be accessed by all CPUs of a multiprocessor system.

Access to Userspace

At many points in the source code there are pointers labeled `__user`; these are also unknown in userspace programming. The kernel uses them to identify pointers to areas in user address space that may not be de-referenced without further precautions. This is because memory is mapped via page tables into the userspace portion of the virtual address space and not directly mapped by physical memory. Therefore the kernel needs to ensure that the page frame in RAM that backs the destination is actually *present* — I discuss this in further detail in Chapter 2. Explicit labeling supports the use of an automatic checker tool (`sparse`) to ensure that this requirement is observed in practice.

1.3.16 . . . and Beyond the Infinite

Although a wide range of topics are covered in this book, they inevitably just represent a portion of what Linux is capable of: It is simply impossible to discuss all aspects of the kernel in detail. I have tried to choose topics that are likely to be most interesting for a general audience and also present a representative cross-section of the whole kernel ecosystem.

Besides going through many important parts of the kernel, one of my concerns is also to equip you with the general idea of why the kernel is designed as it is, and how design decisions are made by interacting developers. Besides a discussion of numerous fields that are not directly related to the kernel (e.g., how the GNU C compiler works), but that support kernel development as such, I have also included a discussion about some nontechnical but social aspects of kernel development in Appendix F.

Finally, please note Figure 1-14, which shows the growth of the kernel sources during the last couple of years.

Kernel development is a highly dynamical process, and the speed at which the kernel acquires new features and continues to improve is sometimes nothing short of miraculous. As a study by the Linux Foundation has shown [KHCM], roughly 10,000 patches go into each kernel release, and this massive

amount of code is created by nearly 1,000 developers per release. On average, 2.83 changes are integrated *every* hour, 24 hours a day, and 7 days a week! This can only be handled with mature means of source code management and communication between developers; I come back to these issues in Appendices B and F.

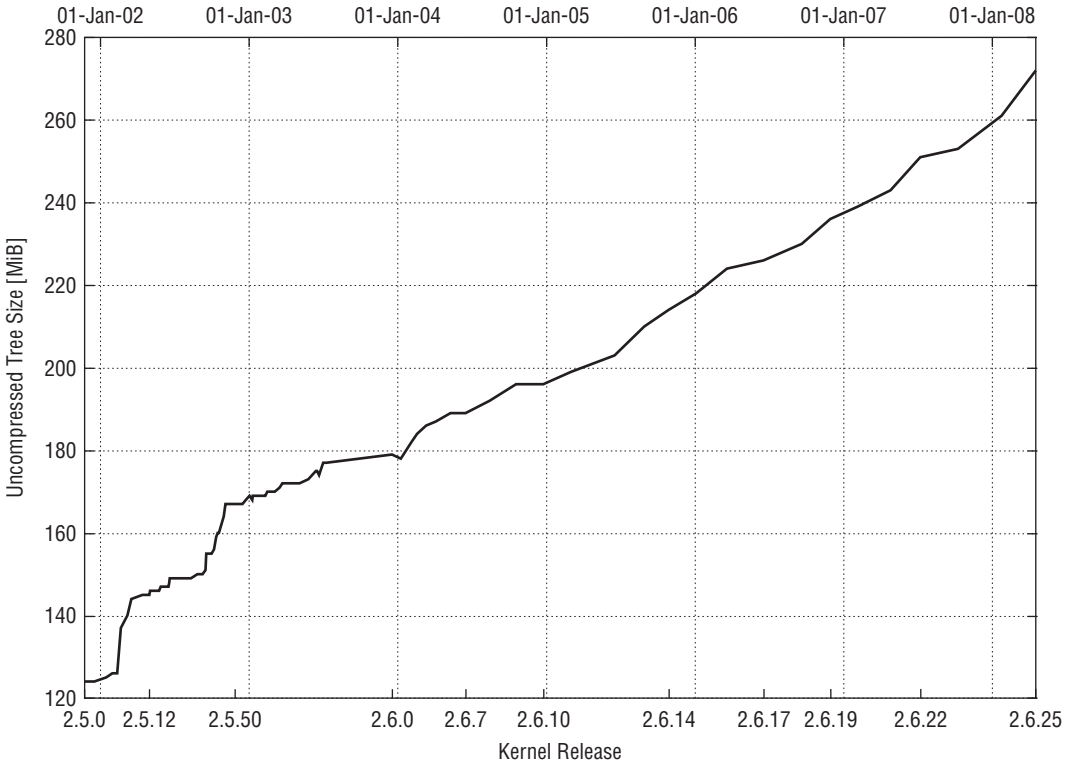

Figure 1-14: Evolution of the core kernel distribution's size during the last years.

1.4 Why the Kernel Is Special

The kernel is an amazing place — but after all, it is just a big C program with some assembler portions (and a drop or two of black magic added sometimes). So what makes the kernel so fascinating? Several factors contribute to this. First and foremost, the kernel is written by the best programmers in the world, and this shows in the code. It is well structured, written with meticulous attention for detail, and contains clever solutions all over the place. In one word: It is code as it ought to be. This, however, does not mean that the kernel is the product of a textbook-style programming methodology: While it employs cleanly designed abstractions to keep the code modular and manageable, it's the mix with the other face of the kernel that makes the code so interesting and unique: If it need be, the kernel does not back off from reusing bit positions in a context-dependent manner, overloading structure elements multiple times, squeezing yet another storage bit out of the aligned portion of pointers, using `goto`s freely, and numerous other things that would make any structured programmer scream miserably in agony and pain.

Techniques that would be unthinkable in many textbook solutions *can* not only be good, but are simply required for a proper real-world working kernel. It's the small path that keeps the balance between these totally opposite faces of the kernel that makes the whole thing so interesting, challenging, and fun!

Having praised the kernel sources, there are a number of more sober issues distinct from userland programs:

- Debugging the kernel is usually harder than debugging userland programs. While for the latter a multitude of debuggers exist, this is much harder to realize for the kernel. There *are* various mechanisms to employ debuggers in kernel development as discussed in Appendix B, but these require more effort than their userland counterparts.
- The kernel provides numerous auxiliary functions that resemble the standard C library found in userspace, but things are much more frugal in the kernel domain.
- Errors in userland applications lead to a segmentation fault or a core dump, but kernel errors will take the whole system down. Or, what is even worse: They will keep the kernel happily running, but manifest themselves in a weird system crash hours after the error occurred. Because debugging in kernel space is harder than for userland applications as mentioned above, it is essential that kernel code receives more thought and judicious consideration than userland code before it is brought into use.
- It must be taken into account that many architectures on which the kernel runs do not support unaligned memory access without further ado. This also affects portability of data structures across architectures because of padding that is inserted by the compiler. This issue is discussed further in Appendix C.
- All kernel code must be protected against concurrency. Owing to the support of multiprocessor machines, Linux kernel code must be both reentrant and thread-safe. That is, routines must allow being executed simultaneously, and data must be protected against parallel access.
- Kernel code must work both on machines with little and big endianness.
- Most architectures do not allow performing floating-point calculations in the kernel without further ado, so you need to find a way to do your calculations with integer types.

You will see how to deal with these issues in the further course of this book.

1.5 Some Notes on Presentation

Before we start to dive right into the kernel, I need to make some remarks on how I present the material, and why I have chosen my particular approach.

Notice that this book is specifically about *understanding* the kernel. Examples of how to write code have intentionally and explicitly been left out, considering that this book is already very comprehensive and voluminous. The works by Corbet et al. [CRKH05], Venkateswaran [Ven08], and Quade/Kunst [QK06] fill in this gap and discuss how to create new code, especially for drivers, by countless practical examples. While I discuss how the kernel build system, which is responsible to create a kernel that precisely suits your needs works, I won't discuss the plethora of configuration options in detail, especially because they are mostly concerned with driver configuration. However, the book by Kroah-Hartman [KH07] can be a valuable aid here.

Usually I start my discussion with a general overview about the concepts of the topic that I am going to present, and then go down to data structures and their interrelation in the kernel. Code is usually discussed last, because this requires the highest level of detail. I have chosen this *top-down* approach because it is in our opinion the most accessible and easiest way to understand the kernel. Notice that it would also be possible to discuss things from the bottom up, that is, start deep down in the kernel and then work slowly up to the C library and userspace level. Notice, however, that presenting something in inverse order does not automatically make it *better*. In my experience, more forward references are required for a bottom-up than for a top-down strategy, so I stick to the latter throughout this book.

When I directly present C source code, I sometimes take the liberty to rewrite it slightly to highlight more important elements and remove less important ''due diligence'' work. For example, it is very important for the kernel to check the return value of every memory allocation. While allocations *will* succeed in nearly almost all cases, it is essential to take care of cases in which not enough memory is available for a particular task. The kernel has to deal with this situation somehow, usually by returning an error return code to userspace if a task is performed as a response to a request by an application, or by omitting a warning message to the system log. However, details of this kind will in general obstruct the view of what is really important. Consider the following code, which sets up namespaces for a process:

kernel/nsproxy.c

```
static struct nsproxy *create_new_namespaces(unsigned long flags,
                        struct task_struct *tsk, struct fs_struct *new_fs)
{
        struct nsproxy *new_nsp;
        int err;

        new_nsp = clone_nsproxy(tsk->nsproxy);
        if (!new_nsp)
                return ERR_PTR(-ENOMEM);

        new_nsp->mnt_ns = copy_mnt_ns(flags, tsk->nsproxy->mnt_ns, new_fs);
        if (IS_ERR(new_nsp->mnt_ns)) {
                err = PTR_ERR(new_nsp->mnt_ns);
                goto out_ns;
        }

        new_nsp->uts_ns = copy_utsname(flags, tsk->nsproxy->uts_ns);
        if (IS_ERR(new_nsp->uts_ns)) {
                err = PTR_ERR(new_nsp->uts_ns);
                goto out_uts;
        }

        new_nsp->ipc_ns = copy_ipcs(flags, tsk->nsproxy->ipc_ns);
        if (IS_ERR(new_nsp->ipc_ns)) {
                err = PTR_ERR(new_nsp->ipc_ns);
                goto out_ipc;
        }
...
        return new_nsp;
out_ipc:
       if (new_nsp->uts_ns)
                put_uts_ns(new_nsp->uts_ns);
out_uts:
       if (new_nsp->mnt_ns)
```

```
                put_mnt_ns(new_nsp->mnt_ns);
out_ns:
        kmem_cache_free(nsproxy_cachep, new_nsp);
        return ERR_PTR(err);
}
```

What the code does in detail is irrelevant right now; I come back to this in the following chapter. What is essential is that the routine tries to clone various parts of the namespace depending on some flags that control the cloning operation. Each type of namespace is handled in a separate function, for instance, in `copy_mnt_ns` for the filesystem namespace.

Each time the kernel copies a namespace, errors can occur, and these must be detected and passed on to the calling function. Either the error is detected directly by the return code of a function, as for `clone_nsproxy`, or the error is encoded in a pointer return value, which can be detected using the `ERR_PTR` macro, which allows for decoding the error value (I also discuss this mechanism below). In many cases, it is not sufficient to just detect an error and return this information to the caller. It is also essential that previously allocated resources that are not required anymore because of the error are released again. The standard technique of the kernel to handle this situation is as follows: Jump to a special label and free all previously allocated resources, or put down references to objects by decrementing the reference count. Handling such cases as this is one of the valid applications for the `goto` statement. There are various possibilities to describe what is going on in the function:

- Talk the reader directly through the code in huge step-by-step lists:

  1. `create_new_namespace` calls `clone_nsproxy`. If this fails, return `-ENOMEM`; otherwise, continue.
  2. `create_new_namespace` then calls `copy_mnt_ns`. If this fails, obtain the error value encoded in the return value of `copy_mnt_ns` and jump to the label `out_ns`; otherwise, proceed.
  3. `create_new_namespace` then calls `copy_utsname`. If this fails, obtain the error value encoded in the return value of `copy_utsname` and jump to the label `out_ns`; otherwise, proceed.
  4. ...

  While this approach is favored by a number of kernel texts, it conveys only little information in addition to what is directly visible from the source code anyway. It is appropriate to discuss some of the most complicated low-level parts of the kernel this way, but this will foster an understanding of neither the big picture in general nor the code snippet involved in particular.

- Summarize what the function does with words, for instance, by remarking that ''`create_new_namespaces` is responsible to create copies or clones of the parent namespaces.'' We use this approach for less important tasks of the kernel that need to be done somehow, but do not provide any specific insights or use particularly interesting tricks.

- Use a flow diagram to illustrate what is going on in a function. With more than 150 code flow diagrams in this book, this is one of my preferred ways of dealing with code. It is important to note that these diagrams *are not supposed* to be a completely faithful representation of the operation. This would hardly simplify matters. Consider Figure 1-15, which illustrates how a faithful representation of `copy_namespaces` could look. It is not at all simpler to read than the source itself, so there is not much purpose in providing it.

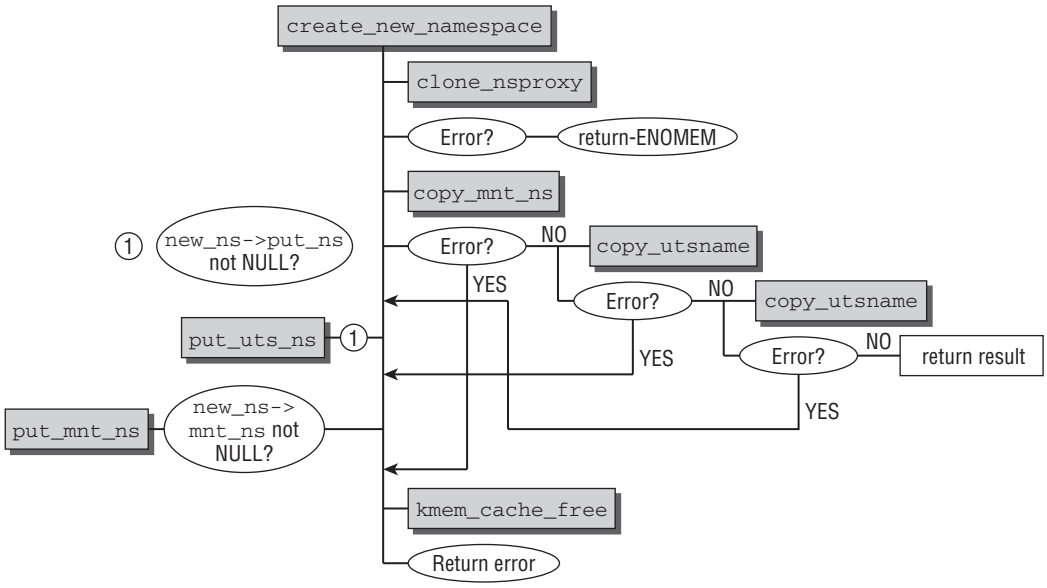

Figure 1-15: Example of a faithful, but unclear and convoluted code flow diagram.

Instead I employ code flow diagrams that illustrate the essential tasks performed by a function. Figure 1-16 shows the code flow diagram that I would have employed instead of Figure 1-15.

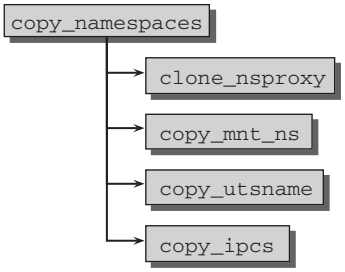

Figure 1-16: Example of the style of code flow diagrams used in this book. They allow immediately catching all essential actions without being distracted by nonessential standard tasks.

The diagram omits several things, but this is on purpose, and also essential. By looking at the figure, you will not see every detail of the function implementation, but you will instead immediately realize that the kernel uses a specific routine to create a clone of each namespace, and the function names provide a sufficient hint of which namespace is copied. This is much more important!

Handling error return codes is something that we assume goes without saying, and so we will not pay particular attention to it. This does not imply that it is not an important thing to do, and in fact it is: Linux would be a lousy kernel if it did not handle this issue properly. But handling errors also obfuscates most operations without introducing any new insights, and does not make it easier to understand the general principles of the kernel, so it's usually better to sacrifice some thoroughness for clarity. The kernel sources are always available for all the gory details!

- It is also often important to discuss kernel code directly if it is packed with important decisions, and I do so when I deem it necessary. However, I often take the liberty of omitting less interesting or purely mechanical parts, so don't be astonished if the code presented in the book sometimes differs slightly from the code seen in the kernel.

With respect to the source code, this book *is* self-contained, but it certainly helps if it is not read on a desolate island, but next to a computer where the Linux source code is available and can be inspected. Besides that, being on a desolate island is not much fun anyway.

Since I base many machine-specific examples on IA-32 and AMD64, some words about these terms are in order. ''IA-32'' includes all Intel-compatible CPUs such as Pentium, Athlon, and so on. AMD64 also includes the Intel variant EM64T. For the sake of simplicity, I use only the abbreviations IA-32 and AMD64 in this book. Since Intel undoubtedly invented IA-32 and AMD came up first with the 64-bit extensions, this seems a fair compromise. It is also interesting to note that starting with kernel 2.6.23, both architectures are unified to the generic x86 architecture within the Linux kernel. This makes the code easier to maintain for the developers because many elements can be shared between both variants, but nevertheless still distinguishes between 32- and 64-bit capabilities of the processors.

1.6 Summary

The Linux kernel is one of the most interesting and fascinating pieces of software ever written, and I hope this chapter has succeeded in whetting your appetite for the things to come in the following chapters, where I discuss many subsystems in detail. For now, I have provided a bird's eye view of the kernel to present the big picture of how responsibilities are distributed, which part of the kernel has to deal with which problems, and how the components interact with each other.

Since the kernel is a huge system, there are some issues related to the presentation of the complex material, and I have introduced you to the particular approach chosen for this book.